# 협동로봇3 프로젝트 참조

## 자율주행 및 협동로봇 관제 시스템


# 협동 로봇 프로젝트 구현


## 시스템

1. 적절한 시나리오 설계
    1. 흥미로운 개발 환경 상세 설계
    2. 개발에서 얻을 수 있는 새로운 지식 탐색
2. 시스템 아키텍처 설계
    1. 하드웨어 구성(로봇 플랫폼, 센서, 제어기)
    2. 소프트웨어 아키텍처(ROS2 노드, DB, GUI 구조)
3. 로봇 모델링 및 하드웨어 플러그인통합
    1. 센서 인터페이스 구현(LiDAR, 카메라, IMU)
4. 지도 생성 및 관리
    1. SLAM을 통한 환경 매핑
    2. 2대 이상의 이동로봇의 협동 로봇 네이게이션 구현
5. 시뮬레이션 월드 모델링
    1. 가제보 환경에서 구현, RVIZ2에서 데이터 모니터링
6. 경로 계획, 이상 환경 탐색 및 장애물 회피
    1. 센서를 활용한 정찰 활동 계획, 상호 협동 시나리오, 관제 시스템 보고
    2. 메니퓰레이터를 활용한 적절한 조취
7. 모니터링 관리 시스템
    1. 위치 추정 및 센서 데이터 모니터링
    2. 원격 제어 기능 구현
8. 데이터 관리 및 로그

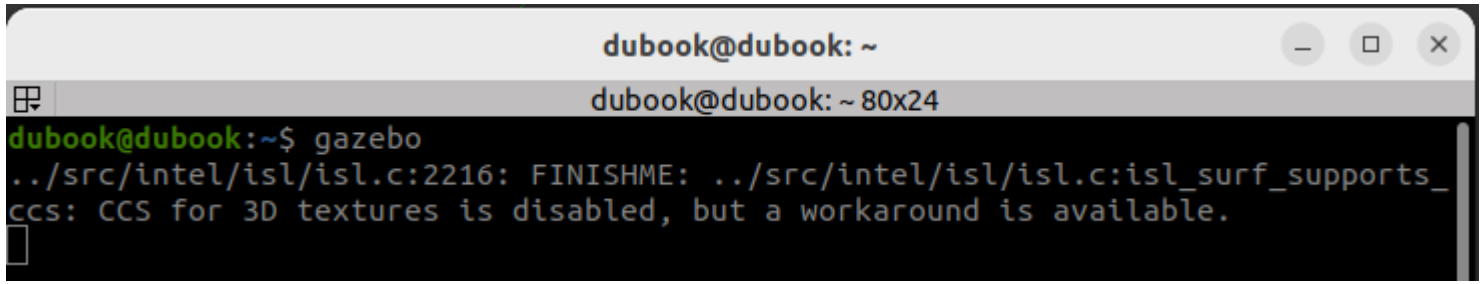
Turtlebot3의 URDF를 수정

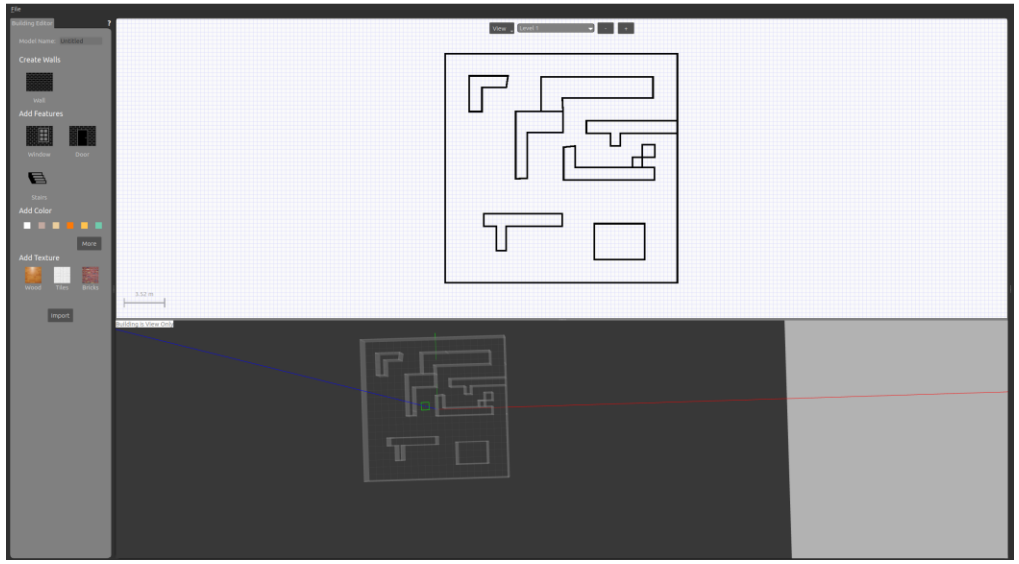
~/turtlebot3_ws/src/turtlebot3_description

1. 주로 다음 경로에 있습니다:
•/urdf/turtlebot3_burger.urdf
•/urdf/turtlebot3_waffle.urdf
•/urdf/turtlebot3_waffle_pi.urdf

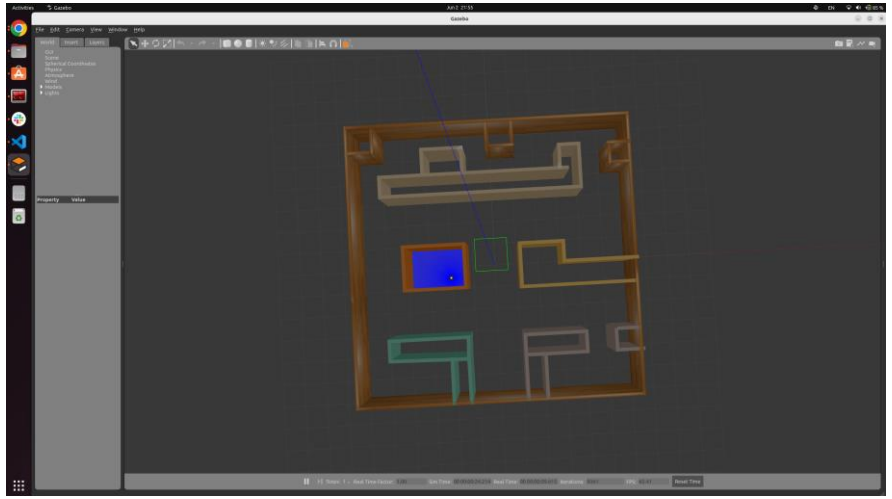
2.수정하는 방법:
2. 원본 패키지를 직접 수정하는 것은 권장되지 않음
3. 대신 새로운 패키지를 만들어서 URDF를 복사하고 수정하는 것을 추천

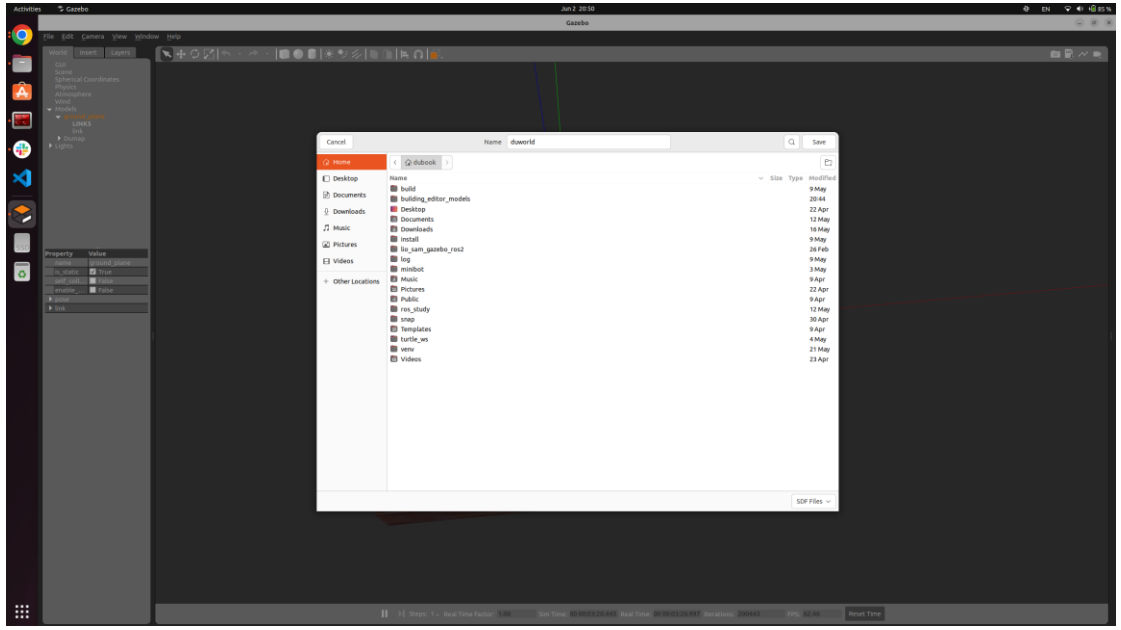
# 터틀봇3에서 시작해보기



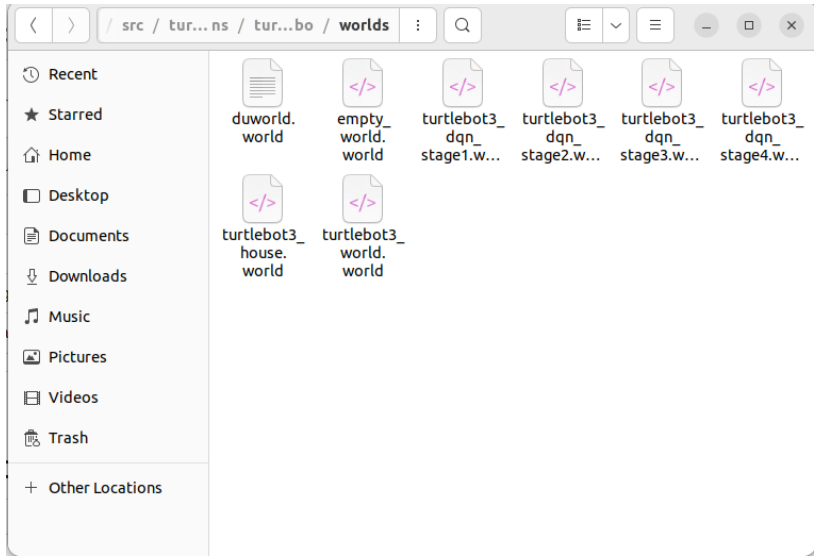
# 터틀봇3에서 시작해보기

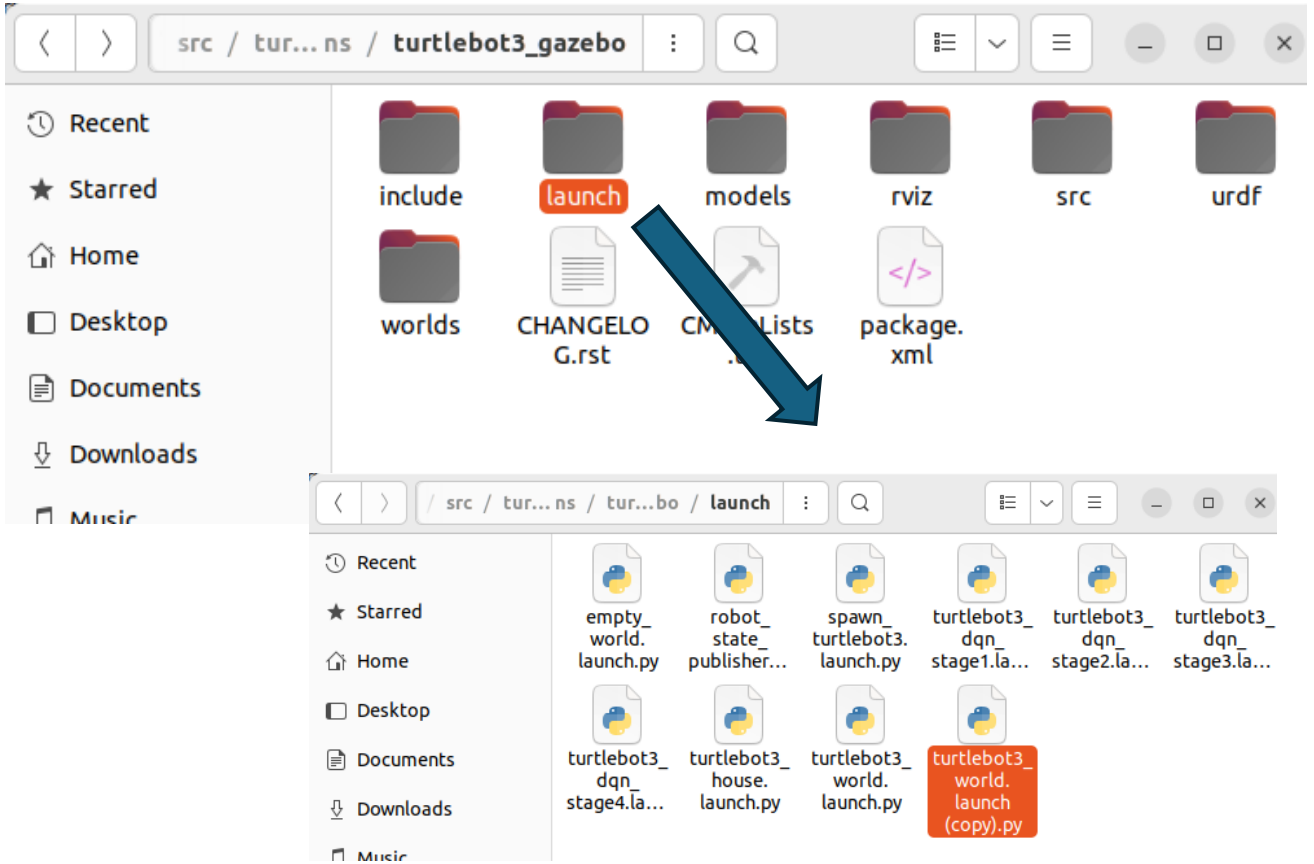




# 터틀봇3에서 시작해보기

방금 만들었던 world파일을 turtlebot3_simulations 패키지 내부의 turtlebot3_gazebo 폴더의 worlds폴더에 넣습니다. 이제 빌드를 하고 나서 실행을 해봅시다.



## 터틀봇3에서 시작해보기

```
world = os.path.join(
    get_package_share_directory('turtlebot3_gazebo'),
    'worlds',
    'duworld.world'
)
```

월드 파일 수정

```
31
32  use_sim_time = LaunchConfiguration('use_sim_time', default='true')
33  x_pose = LaunchConfiguration('x_pose', default='-2.0')
34  y_pose = LaunchConfiguration('y_pose', default='-0.5')
```

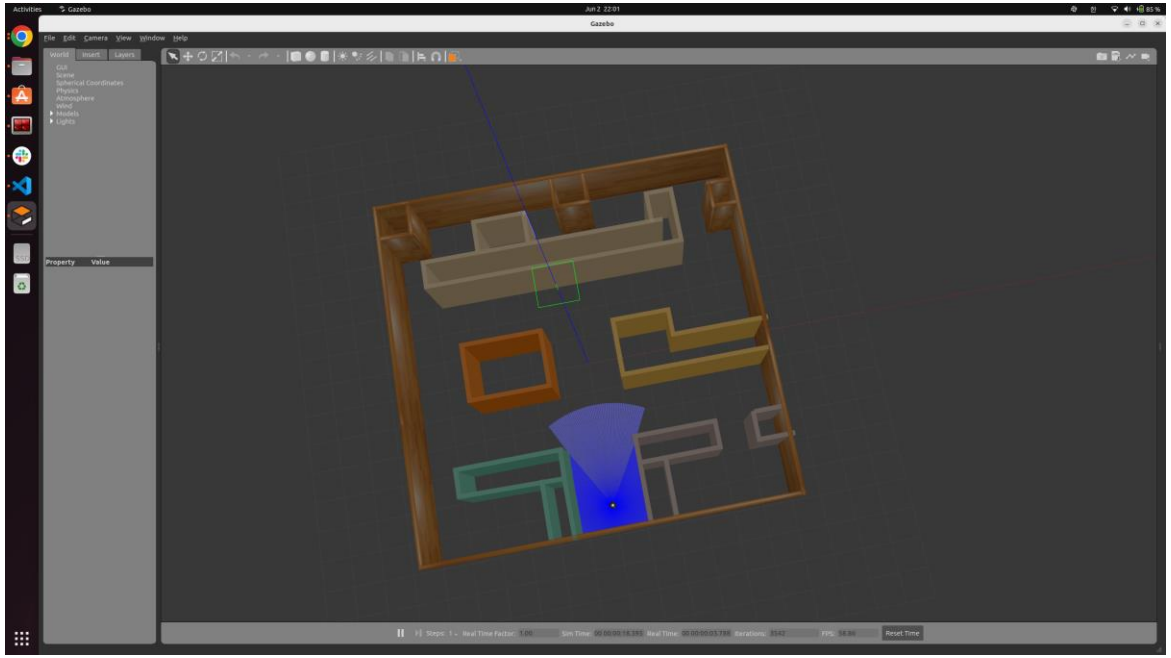
초기 위치 수정

```
dubook@dubook:~$ turtlebot
ROS2 Humble is activated.
turtlebot workspace is activated.
dubook@dubook:~$ cd turtle_ws/
dubook@dubook:~/turtle_ws$ colcon build
Starting >>> turtlebot3_msgs
Starting >>> dynamixel_sdk
Starting >>> turtlebot3_description
Starting >>> dynamixel_sdk_custom_interfaces
Starting >>> turtlebot3_cartographer
Starting >>> turtlebot3_gazebo
Starting >>> turtlebot3_navigation2
Starting >>> turtlebot3_teleop
Finished <<< turtlebot3_cartographer [0.34s]
Finished <<< turtlebot3_navigation2 [0.35s]
Finished <<< turtlebot3_description [0.43s]
Finished <<< dynamixel_sdk [0.45s]
```

빌드 및 소싱

```
ros2 launch turtlebot3_gazebo turtlebot3_duworld.launch.py
```

# 터틀봇3에서 시작해보기



SLAM

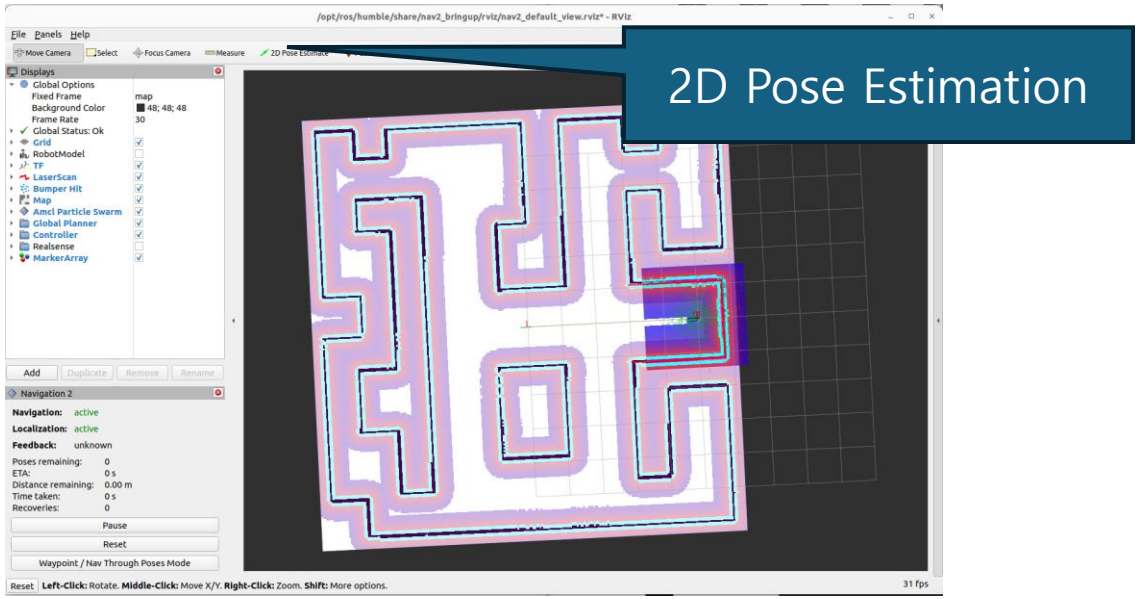
```
ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
ros2 run turtlebot3_teleop teleop_keyboard
ros2 run nav2_map_server map_saver_cli -f ~/map
```

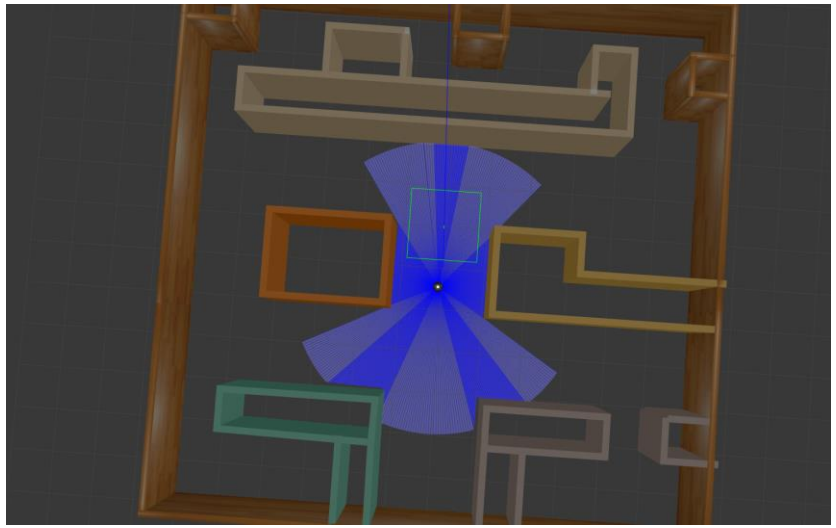
# 터틀봇3에서 시작해보기

Navigation

```
ros2 launch turtlebot3_gazebo turtlebot3_duworld.launch.py

ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True
map:=$HOME/map.yaml
```




# 협동 로봇 프로젝트 참고

# Waypoint Navigation Node

```
import rclpy
from rclpy.node import Node
from rclpy.action import ActionClient
from geometry_msgs.msg import PoseStamped
from nav2_msgs.action import NavigateToPose
from rclpy.duration import Duration
import math

class WaypointNavigator(Node):
    def __init__(self):
        super().__init__('waypoint_navigator')

        # 웨이포인트 리스트 [(x, y, theta), ...]
        self.waypoints = [
            (1.0, 1.0, 0.0),     # 웨이포인트 1
            (2.0, 2.0, 1.57),    # 웨이포인트 2
            (2.0, 4.0, 3.14),    # 웨이포인트 3
            (0.0, 4.0, -1.57),   # 웨이포인트 4
            (0.0, 0.0, 0.0),     # 웨이포인트 5
        ]

        self.current_waypoint = 0

        # Nav2 액션 클라이언트 생성
        self.nav_client = ActionClient(self, NavigateToPose, 'navigate_to_pose')

        # 웨이포인트 도달 허용 오차
        self.position_tolerance = 0.2  # meters
        self.orientation_tolerance = 0.1  # radians

        # 타이머 생성 (2초마다 상태 체크)
        self.create_timer(2.0, self.navigation_callback)

        self.get_logger().info('Waypoint Navigator has been initialized')

    def navigate_to_waypoint(self, x, y, theta):
        """지정된 웨이포인트로 이동하는 함수"""
        # Nav2 목표점 메시지 생성
        goal = NavigateToPose.Goal()
        goal.pose.header.frame_id = 'map'
        goal.pose.header.stamp =
self.get_clock().now().to_msg()

        # 위치 설정
        goal.pose.pose.position.x = x
        goal.pose.pose.position.y = y
        goal.pose.pose.position.z = 0.0

        # 방향을 쿼터니언으로 변환하여 설정
        q = self.euler_to_quaternion(0.0, 0.0, theta)
        goal.pose.pose.orientation.x = q[0]
        goal.pose.pose.orientation.y = q[1]
        goal.pose.pose.orientation.z = q[2]
        goal.pose.pose.orientation.w = q[3]

        # 네비게이션 목표 전송
        self.nav_client.wait_for_server()
        self.future = self.nav_client.send_goal_async(goal)

self.future.add_done_callback(self.goal_response_callback)

def goal_response_callback(self, future):
    """네비게이션 목표 응답 처리 콜백"""
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().error('Goal rejected')
        return

    self.get_logger().info('Goal accepted')
    self._get_result_future = goal_handle.get_result_async()
    self._get_result_future.add_done_callback(self.get_result_callback)

def get_result_callback(self, future):
    """네비게이션 결과 처리 콜백"""
    status = future.result().status
    if status == 4:  # 성공적으로 도달
        self.get_logger().info(f'Reached waypoint {self.current_waypoint}')
        self.current_waypoint += 1
        if self.current_waypoint >= len(self.waypoints):
            self.get_logger().info('All waypoints reached!')
            self.current_waypoint = 0  # 처음부터 다시 시작 (선택사항)
    else:
        self.get_logger().error(f'Navigation failed with status: {status}')
```

## Waypoint Navigation Node

```
def navigation_callback(self):
    """주기적으로 실행되는 네비게이션 콜백"""
    if not hasattr(self, 'future') or self.future.done():
        if self.current_waypoint < len(self.waypoints):
            wp = self.waypoints[self.current_waypoint]
            self.get_logger().info(f'Navigating to waypoint {self.current_waypoint}: {wp}')
            self.navigate_to_waypoint(*wp)

def euler_to_quaternion(self, roll, pitch, yaw):
    """오일러 각을 쿼터니언으로 변환"""
    cy = math.cos(yaw * 0.5)
    sy = math.sin(yaw * 0.5)
    cp = math.cos(pitch * 0.5)
    sp = math.sin(pitch * 0.5)
    cr = math.cos(roll * 0.5)
    sr = math.sin(roll * 0.5)

    q = [0] * 4
    q[0] = sr * cp * cy - cr * sp * sy
    q[1] = cr * sp * cy + sr * cp * sy
    q[2] = cr * cp * sy - sr * sp * cy
    q[3] = cr * cp * cy + sr * sp * sy

    return q
```

```
def main(args=None):
    rclpy.init(args=args)
    navigator = WaypointNavigator()
    rclpy.spin(navigator)
    navigator.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

# 장애물 추가

노드 제작 후 spawn_entity 서비스 사용

```python
def __init__(self):
    super().__init__('random_box_spawner')

    self.client = self.create_client(SpawnEntity, '/spawn_entity')

    while not self.client.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('Waiting for spawn_entity service...')

    self.get_logger().info('SpawnEntity service available!')
```

```python
def spawn_random_boxes(self):
    # 색깔별 상자 모델 이름
    colors = ["box_red", "box_green", "box_blue", "box_yellow"]
    random.shuffle(colors)  # 리스트 셔플

    # 상자의 위치 설정 (순서대로 위치를 다르게 배치)
    positions = [
        (0.000753, 7.300000, 1.082375),  # 첫 번째 상자의 위치
        (0.000753, 2.586649, 1.082375),  # 두 번째 상자의 위치
        (0.000753, -1.413351, 1.082375),  # 세 번째 상자의 위치
        (0.000753, -5.413351, 1.082375)   # 네 번째 상자의 위치
    ]

    # 셔플된 색상을 순서대로 배치
    for box_id, (color, (x, y, z)) in enumerate(zip(colors, positions)):
        self.spawn_box(color, box_id, x, y, z)
```

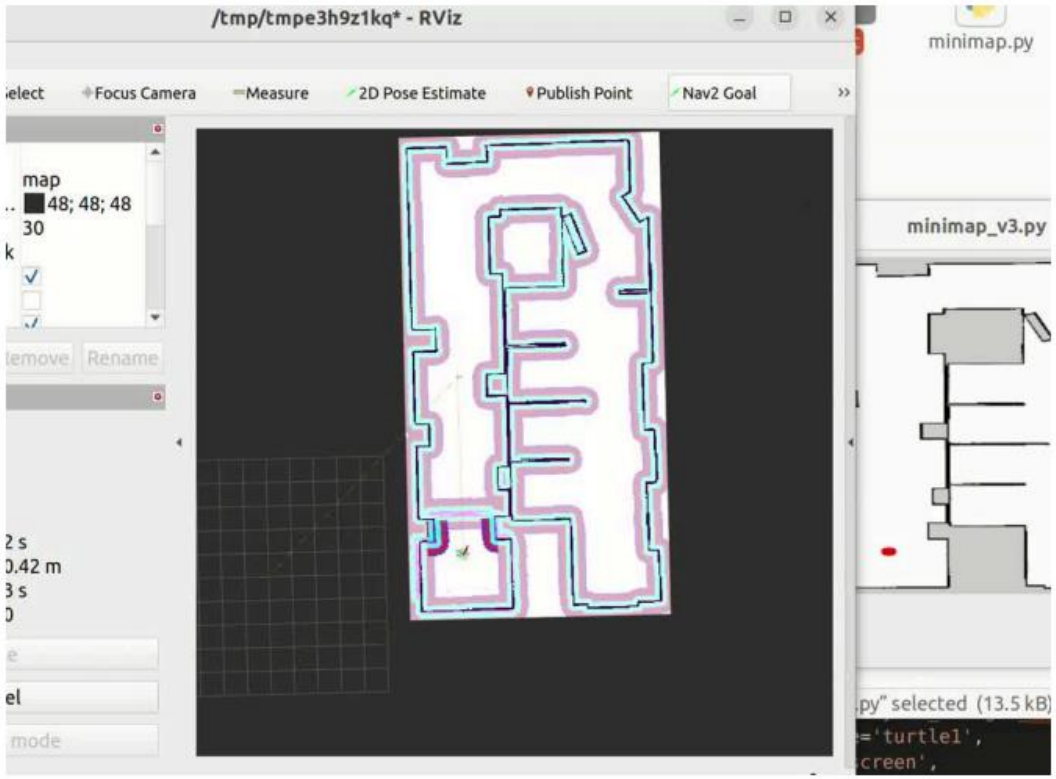
제작한 벽 모델 스폰 로직

스폰 서비스 call

```python
def spawn_box(self, selected_color, box_id, x, y, z):
    # Gazebo SpawnEntity 서비스 요청 생성
    request = SpawnEntity.Request()
    request.name = f"{selected_color}_{box_id}"  # 각 상자의 고유 이름 지정
    request.xml = f"""
    <sdf version="1.6">
      <model name="{selected_color}">
        <include>
          <uri>model://{selected_color}</uri>
        </include>
      </model>
    </sdf>
    """
    request.robot_namespace = ''
    request.initial_pose.position.x = x
    request.initial_pose.position.y = y
    request.initial_pose.position.z = z

    # 서비스 호출
    future = self.client.call_async(request)
    rclpy.spin_until_future_complete(self, future)
    if future.result() is not None:
        self.get_logger().info(f"Successfully spawned {selected_color} at ({x
    else:
        self.get_logger().error("Failed to spawn box")
```

# 실시간 위치 이미지위에 표시




## ROBOT1 위치 실시간 확인

```python
def process_signal(self, message):
    # AMCL 좌표 수신
    self.amcl_pose_x = message[0]  # 인스턴스 변수로 저장
    self.amcl_pose_y = message[1]  # 인스턴스 변수로 저장

    # 맵 좌표를 픽셀 좌표로 변환 (RViz 좌표계와 GUI 좌표계 동기화)
    pixel_x = int((self.map_x + self.amcl_pose_x) / self.resolution)
    pixel_y = int(self.height - ((self.map_y + self.amcl_pose_y) / self.resolution))

    # 맵 이미지 범위 내로 클리핑
    pixel_x = max(0, min(self.width - 1, pixel_x))
    pixel_y = max(0, min(self.height - 1, pixel_y))
```

'/turtle1/amcl_pose' 구독으로 x,y 좌표 수신

```python
# 이미지 회전: 90도 반시계 방향 회전
rotated_image = cv2.rotate(image, cv2.ROTATE_90_COUNTERCLOCKWISE)

# Grayscale -> BGR로 변환
rotated_image_color = cv2.cvtColor(rotated_image, cv2.COLOR_GRAY2BGR)

# 로봇 좌표를 픽셀 좌표로 변환
try:
    # 좌표 변환 공식 (amcl_pose -> 픽셀 좌표)
    pixel_x = int((self.amcl_pose_y - (-17.0)) / self.resolution)
    pixel_y = int((self.amcl_pose_x - (-2.0)) / self.resolution)

    # 좌-우 반전 보정
    corrected_pixel_x = rotated_image_color.shape[1] - pixel_x  # 이미지 너비에서 x를 반전
    corrected_pixel_y = rotated_image_color.shape[0] - pixel_y  # 이미지 높이에서 y 조정
```

map.pgm이 가로지만 rviz에서 세로이기 때문에 90반시계 회전

## 엔터티 제거

```python
self.delete_client = self.create_client(
    DeleteEntity,
    '/delete_entity',
    callback_group=self.callback_group
)

while not self.delete_client.wait_for_service(timeout_sec=2.0)
    self.get_logger().info('Delete Service not available, wait
```

벽의 제거는 delete_entity 서비스를 이용하였다.

벽 제거 서비스 call >

```python
### 벽 제거 ###
def delete_box(self, box_name):
    request = DeleteEntity.Request()
    request.name = f'{box_name}_{self.num}'
    future = self.delete_client.call_async(request)

    # Add a done callback to handle the response asynchronously
    future.add_done_callback(self.delete_box_callback)

def delete_box_callback(self, future):
    try:
        result = future.result()
        if result is not None:
            self.get_logger().info(f"Wall is deleted successfully")
        else:
            self.get_logger().error("Failed to delete box")
    except Exception as e:
        self.get_logger().error(f"Error while deleting box: {str(e)}")
```

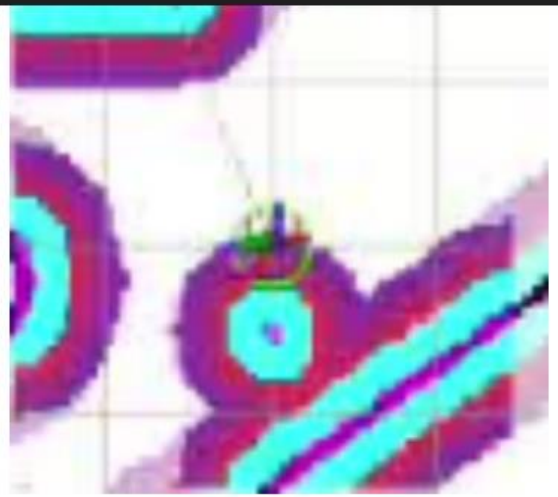
## 로봇이 모두 스폰된 이후에 rviz와 Drive 노드 실행

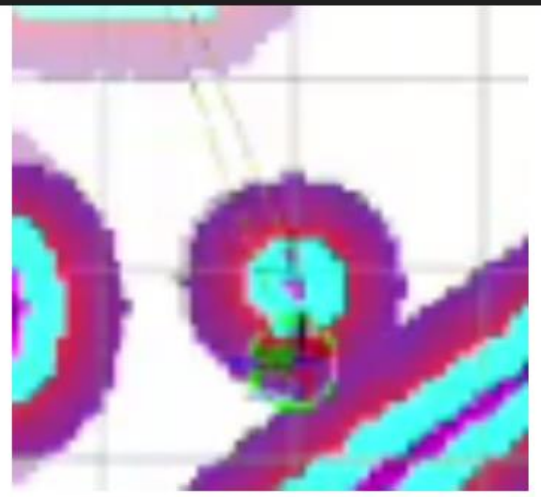
- turtlebot3_multi_robot **(gazebo, rviz)**

```
# Start rviz nodes and drive nodes after the last robot is spawned
for robot in robots:

    namespace = [ '/' + robot['name'] ]

    # Create a initial pose topic publish call
    if namespace[0] == '/tb1':
        message = '{header: {frame_id: map}, pose: {pose: {position: {x: 0.0, y: 0.0, z: 0.1}, orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}, }
    else:
        message = '{header: {frame_id: map}, pose: {pose: {position: {x: -0.5, y: 0.0, z: 0.1}, orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}},
```






## 네임스페이스를 추가하여 NavigateToPose 액션 서버와 연결

```python
class MoveToGoal(Node):
    def __init__(self, namespace=''):
        super().__init__('move_to_goal_node')

        # 네임스페이스를 추가하여 NavigateToPose 액션 서버에 연결
        self._client = ActionClient(
            self,
            NavigateToPose,
            f'{namespace}/navigate_to_pose'  # 네임스페이스를 동적으로 할당
        )
        self.namespace = namespace
        self.current_goal_complete = False
        self.start_mission = False
        self.stop_requested = False
```


## 시스템

/amcl_pose (AMCL - Localization)

퍼블리시하는 노드: amcl (nav2_amcl)

네비게이션을 실행하면 AMCL(Adaptive Monte Carlo Localization)이 로봇의 위치를 추정하여 /amcl_pose 토픽으로 퍼블리시

/pose (Cartographer SLAM - Mapping)

퍼블리시하는 노드: cartographer_node

Cartographer SLAM을 사용할 경우, 이 노드가 로봇의 현재 위치를 추정하여 /pose 토픽으로 퍼블리시

/odometry/filtered (EKF Localization - Optional)

퍼블리시하는 노드: ekf_localization_node (robot_localization 패키지 사용 시)

만약 IMU와 오도메트리를 융합하여 정밀한 위치 추정을 수행한다면, EKF 필터가 적용된 위치를 /odometry/filtered에서 제공

## 로봇의 위치 추적(PoseWithCovarianceStamped Message)

**amcl(Adaptive Monte Carlo Localization) 노드는 로봇의 위치를 지도(/map) 좌표계에서 추정하여 퍼블리시**

**nav2_pose 노드에서 발행**

**ros2 topic echo /amcl_pose**

**Nav2 네비게이션을 실행할 때만 동작**

**ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=true map:=/path/to/map.yaml**

```
# 첫 번째 로봇 위치 추적 (tb1/amcl_pose)
if namespace == 'tb2':
    self.leader_pose_x = 0.0
    self.leader_pose_y = 0.0
    self.leader_yaw = 0.0
    self.create_subscription(
        PoseWithCovarianceStamped,
        '/tb1/amcl_pose',
        self.leader_pose_callback,
        10
    )
```

**실행 중인 모든 토픽 확인**

**ros2 topic list**

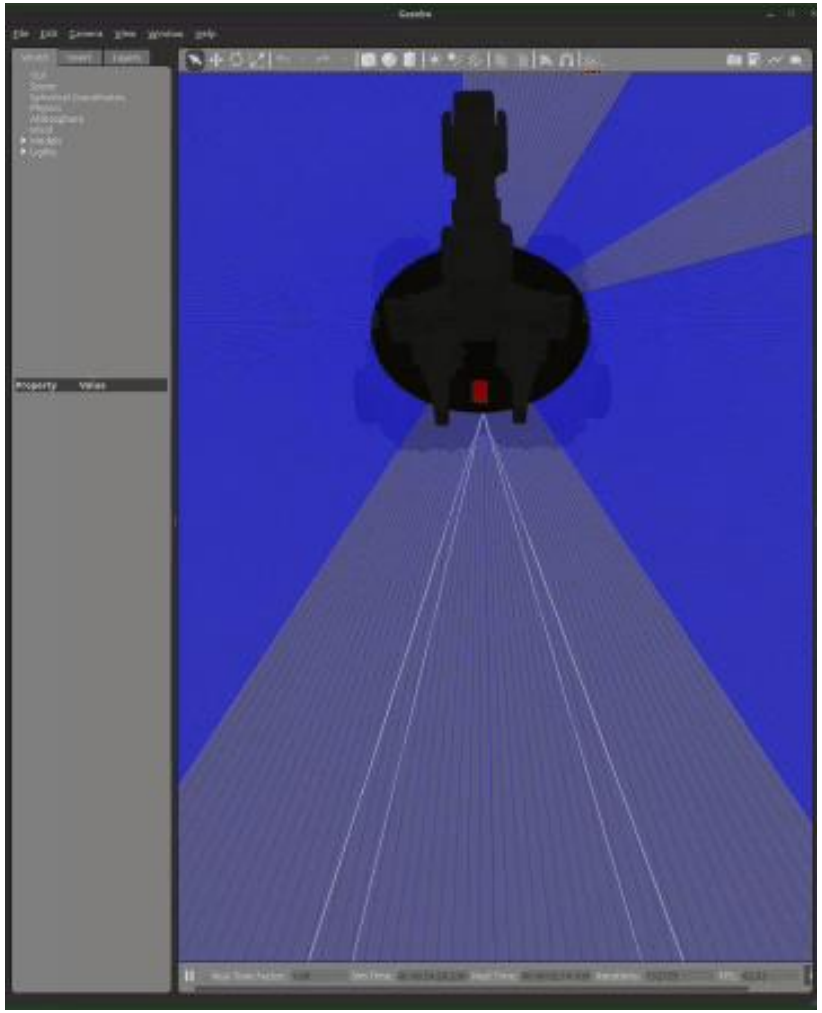
**특정 토필의 퍼블리셔 확인**

**ros2 topic info /amcl_pose**

**메시지 데이터 확인**

**ros2 topic echo /amcl_pose**

# 협동로봇 - manipulator



```python
class OpenManipulatorWithGripper(Node):
    def __init__(self):
        super().__init__('open_manipulator_with_gripper')

        # Initialize publishers and action clients
        self.joint_publisher = self.create_publisher(JointTrajectory, '/arm_controller/joint_trajectory', 10)
        self.gripper_action_client = ActionClient(self, GripperCommand, 'gripper_controller/gripper_cmd')

        # Constants for manipulator arm
        self.r1 = 130
        self.r2 = 124
        self.r3 = 126
        self.th1_offset = -math.atan2(0.024, 0.128)
        self.th2_offset = -0.5 * math.pi - self.th1_offset

        # Initialize trajectory message
        self.trajectory_msg = JointTrajectory()
        self.trajectory_msg.joint_names = ['joint1', 'joint2', 'joint3', 'joint4']
```

```
#xacro 파일 urdf 변환
ros2 run xacro xacro --inorder /home/viator/turtlebot3_ws/src/turtlebot3_manipulation/
turtlebot3_manipulation_description/urdf/turtlebot3_manipulation.urdf.xacro -o /home/viator/
Downloads/multitb_ws/src/turtlebot3_multi_robot/urdf/turtlebot3_manipulation.urdf


#URDF->SDF변환
gz sdf -p turtlebot3_manipulation.urdf > turtlebot3_manipulation.sdf
gz sdf -p turtlebot3_with_basket.urdf > turtlebot3_with_basket.sdf
```

# 협동로봇 - 라이다 검출 물체 추적

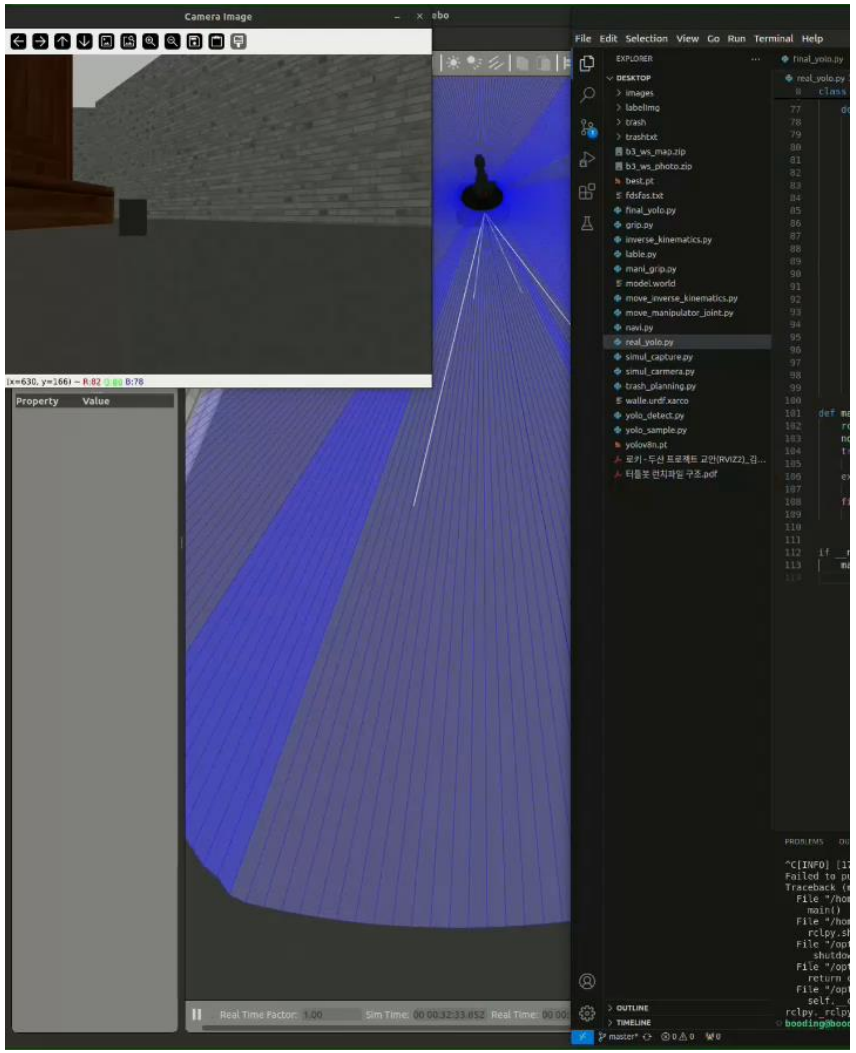
```python
def lidar_callback(self, scan_msg):
    if not self.is_adjusting_distance:
        return  # 거리 조정 중이 아니면 무시

    # LiDAR 데이터 처리
    min_distance = min(scan_msg.ranges)  # LiDAR 데이터에서 최소 거리 추출
    self.get_logger().info(f"Minimum distance to object: {min_distance} m")

    twist = Twist()

    # 거리 조정 로직
    if min_distance > self.target_distance:  # 목표 거리보다 멀면 접근
        self.get_logger().info("Too far. Moving closer.")
        twist.linear.x = 0.1  # 천천히 전진
    elif min_distance < self.target_distance:
        self.get_logger().info("Too close. Moving backward.")
        twist.linear.x = -0.1  #
    else:
        self.get_logger().info("Target distance achieved. Stopping.")
        self.is_adjusting_distance = False  # 거리 조정 완료
        twist.linear.x = 0.0  # 정지

    self.cmd_vel_pub.publish(twist)
```




# 협동로봇 - 카메라 물체 추적

```python
def camera_callback(self, image_msg):
    if not self.is_tracking or self.is_adjusting_distance:
        return  # 트래킹 중이 아니거나 거리 조정 중일 때는 무시

    try:
        # ROS 2 Image -> OpenCV Image
        cv_image = self.bridge.imgmsg_to_cv2(image_msg, "bgr8")

        # YOLOv8 추론
        results = self.model(cv_image, verbose=False)
        boxes = results[0].boxes.data.cpu().numpy()  # numpy 배열로 변환
        self.get_logger().info(f"Detected {len(boxes)} objects.")

        if len(boxes) == 0:
            return  # 감지된 객체가 없으면 처리 중지

        # 첫 번째 객체를 사용 (예: 중앙점 계산)
        x1, y1, x2, y2, conf, cls = boxes[0]
        cx = (x1 + x2) // 2  # 바운딩 박스 중심 x좌표

        # 속도 명령 생성
        twist = Twist()
        twist.linear.x = 0.0  # 트래킹 중에는 선형 이동하지 않음
        image_center = cv_image.shape[1] // 2

        # 방향 계산
        angular_error = (image_center - cx) * 0.001  # 기본 방향 계산
        twist.angular.z = angular_error

        # 정중앙에 위치하면 멈추고 거리 조정 상태로 전환
        if abs(cx - image_center) < 10:  # 10 픽셀 이내
            self.get_logger().info("Object centered. Switching to distance adjustment.")
            self.is_tracking = False  # 트래킹 종료
            self.is_adjusting_distance = True  # 거리 조정 시작
            stop_twist = Twist()
            self.cmd_vel_pub.publish(stop_twist)  # 정지 명령 퍼블리시
            return

        self.get_logger().info(f"Bounding Box Center: {cx}, Image Center: {image_center}, Angular Error: {angular_error}")
        self.get_logger().info(f"Computed Twist - Linear X: {twist.linear.x}, Angular Z: {twist.angular.z}")

        # 이동 명령 저장
        self.current_twist = twist
        self.cmd_vel_pub.publish(twist)

    except Exception as e:
        self.get_logger().error(f"Failed to process image: {e}")
```


# 협동로봇 - 물체로 로봇팔의 위치 변화

```python
def solv_robot_arm2(self, x, y, z, r1, r2, r3):
    """Calculate joint angles for the robot arm to reach the given (x, y, z)."""
    Rt = math.sqrt(x**2 + y**2 + z**2)
    if Rt > (r1 + r2 + r3):
        raise ValueError("Target position is out of the manipulator's reach.")

    St = math.asin(z / Rt)
    Sxy = math.atan2(y, x)
    s1, s2 = self.solv2(r1, r2, Rt)
    sr1 = math.pi / 2 - (s1 + St)
    sr2 = s1 + s2
    sr3 = math.pi - (sr1 + sr2)
    return sr1, sr2, sr3, Sxy

def solv2(self, r1, r2, r3):
    """Calculate angles between links."""
    d1 = (r3**2 - r2**2 + r1**2) / (2 * r3)
    d2 = (r3**2 + r2**2 - r1**2) / (2 * r3)
    s1 = math.acos(d1 / r1)
    s2 = math.acos(d2 / r2)
    return s1, s2
```

```python
def update_position(self, x, y, z):
    """Calculate joint angles and update trajectory message."""
    try:
        sr1, sr2, sr3, Sxy = self.solv_robot_arm2(x, y, z, self.r1, self.r2, self.r3)
        joint_angles = [Sxy, sr1 + self.th1_offset, sr2 + self.th2_offset, sr3]

        # Update trajectory point
        point = JointTrajectoryPoint()
        point.positions = joint_angles
        point.velocities = [0.0] * 4
        point.time_from_start.sec = 3
        self.trajectory_msg.points = [point]

        # Publish trajectory
        self.joint_publisher.publish(self.trajectory_msg)
        self.get_logger().info(f"Published trajectory with joint angles: {joint_angles}")

    except ValueError as e:
        self.get_logger().error(f"Error calculating joint angles: {e}")
```

**gazebo_link_attacher**

**Gazebo 시뮬레이터에서 사용되는 ROS 패키지**

**시뮬레이션 중에 두 개의 링크(물체)를 동적으로 연결할 수 있게 해주는 도구**

```
# 패키지 설치
cd ~/catkin_ws/src
git clone https://github.com/pal-robotics/gazebo_ros_link_attacher.git
cd ..
catkin_make

# 실행 예시
roslaunch gazebo_ros_link_attacher attacher.launch
```


## 협동로봇 – 그리퍼, Pack And Place

- 프락시 서비스 클라이언트 생성 방법

  •Attach는 서비스의 전체 인터페이스를 정의
  •AttachRequest는 요청 데이터의 구조만을 정의
  •실제 사용할 때는 AttachRequest를 생성하여 데이터를 채우고, Attach 서비스를 통해 전송합니다

```
import rospy
from gazebo_ros_link_attacher.srv import Attach, AttachRequest

# 서비스 클라이언트 생성
attach_srv = rospy.ServiceProxy('/link_attacher_node/attach', Attach)
detach_srv = rospy.ServiceProxy('/link_attacher_node/detach', Attach)

# 링크 연결 요청
req = AttachRequest()
req.model_name_1 = "robot"
req.link_name_1 = "gripper_link"
req.model_name_2 = "object"
req.link_name_2 = "link"

# 연결/분리 실행
attach_srv.call(req)  # 링크 연결
detach_srv.call(req)  # 링크 분리
```

- /AttachLink,/ DetachLink 서비스 클라이언트 생성 방법

```
<plugin name='gazebo_link_attacher' filename='libgazebo_link_attacher.so'/>
```

```
ros2 service call /ATTACHLINK linkattacher_msgs/srv/AttachLink
"{model1_name: 'arduinobot', link1_name: 'link_j6e', model2_name: 'tb1', link2_name: 'tb1::base_link'}"

ros2 service call /DETACHLINK linkattacher_msgs/srv/DetachLink
"{model1_name: 'arduinobot', link1_name: 'link_j6e', model2_name: 'tb1', link2_name: 'tb1::base_link'}"
```

```
self.attach_client = self.create_client(AttachLink, "/ATTACHLINK")
self.detach_client = self.create_client(DetachLink, "/DETACHLINK")
```

```
if robot_name is not None:
    time.sleep(3)  # Slow down before attach
    self._attach_then_move(destination_num, robot_name)
```

```
req = AttachLink.Request()
req.model1_name = "arduinobot"
req.link1_name = "link_j6e"
req.model2_name = robot_name
req.link2_name = f"{robot_name}::base_link"

self.get_logger().info(f"[async_attach_object] Sending AttachLink => {req}")
future = self.attach_client.call_async(req)
```


## ServiceProxy()에 대하여

**rospy.ServiceProxy는 ROS에서 서비스 클라이언트를 생성하는 클래스**

**다른 노드가 제공하는 서비스를 호출할 때 사용**

```
import rospy
from std_srvs.srv import SetBool

# 서비스 프록시 생성
service_client = rospy.ServiceProxy('service_name', SetBool)

# 서비스 호출
try:
    response = service_client(True)  # 서비스 요청 데이터 전달
    print(response.success)  # 응답 처리
except rospy.ServiceException as e:
    print("Service call failed: %s" % e)
```


## ServiceProxy() 예시

```
import rospy
from geometry_msgs.srv import GetPlanRequest, GetPlan

def request_path_plan():
    # 서비스 프록시 생성
    plan_service = rospy.ServiceProxy('move_base/make_plan', GetPlan)

    # 서비스가 사용 가능할 때까지 대기
    rospy.wait_for_service('move_base/make_plan')

    # 요청 생성
    req = GetPlanRequest()
    req.start = start_pose
    req.goal = goal_pose

    try:
        # 서비스 호출
        response = plan_service(req)
        return response.plan
    except rospy.ServiceException as e:
        rospy.logerr("Service call failed: %s" % e)
        return None
```

# 감사합니다.